INTRODUCING ITERATION INTO THE PURE LISP THEOREM PROVER

BY J STROTHER MOORE

XEROX
PALO ALTO RESEARCH CENTER

INTRODUCING ITERATION INTO THE PURE LISP THEOREM PROVER

BY J STROTHER MOORE

CSL 74-3 DECEMBER, 1974 (REVISED MARCH, 1975)

It is shown how the LISP iterative primitives PROG, SETQ, GO, and RETURN may be introduced into the Boyer-Moore method for automatically verifying Pure LISP programs. This is done by extending some of the previously described heuristics for dealing with recursive functions. The resulting verification procedure uses structural induction to handle both recursion and iteration. The procedure does not actually distinquish between the two and they may be mixed arbitrarily. For example, since properties are stated in terms of user-defined functions, the theorem prover will prove recursively specified properties of iterative functions. Like its predecessor, the procedure does not require user-supplied inductive assertions for the iterative programs.

KEY WORDS AND PHRASES: LISP, automatic theorem proving, structural inductions, program verification.

CR CATEGORIES: 3.64, 4.22, 5.21.

XEROX
PALO ALTO RESEARCH CENTER
3333 COYOTE HILL ROAD / PALO ALTO / CALIFORNIA 94304

1. BACKGROUND

This paper assumes the reader is familiar with the description of the Boyer-Moore Pure LISP Theorem Prover, as found in [2] or [7]. This section presents an informal summary of the theorem prover as it stood before the current extension.

The theorem prover deals with functions defined recursively in terms of the LISP-like primitives NIL, CONS, CAR, CDR, EQUAL, and an n-place conditional operator called COND[1]. Properties to be proved are stated as LISP expressions. Given such an expression, the theorem prover tries to show that it would always evaluate to T for any values of the free variables in it[2]. The proof procedure is composed of four parts: simplification, which applies equivalence preserving rewrite rules (including those of symbolic evaluation); fertilization, which is an equality substitution rule which "uses" equalities and then throws them away; generalization, which generalizes the theorem to be proved; and a structural induction heuristic.

In this paper we will describe extensions to the generalization and induction heuristics. Therefore, it is important to get a good understanding of them.

Generalization is often necessary because of the presence of induction. Intuitively, to prove a theorem by induction one must show (among other things) that the truth

[1] *The conditional operator in the earlier papers was three-place. However, since* COND *in LISP actually takes an indefinite number of clause arguments, the three-place* COND *was confusing. Therefore, the LISP conditional was adopted.* (COND (p x)(q y) ... (T z)) *is read: if* p *is non-*NIL *then* x, *otherwise if* q *is non-*NIL *then* y, ... *otherwise,* z.

[2] T *is the true truth value and is defined to be the non-*NIL *object* EQUAL *to the value of* (CONS NIL NIL).

of one instance of the expression implies the truth of another (related) instance. This means that it is often necessary to prove a more general theorem than that given, merely because one needs the additional generality in the hypothesis. This is frequently expressed by saying that the original theorem "is not strong enough to carry itself through induction." The generalization heuristic described in [2] and [7] finds non-trivial subterms common to both sides of equalities and implications, and replaces them by new universally quantified variables. For example, if the theorem to be proved was:

```
(EQUAL (APPEND (REVERSE X) NIL) (REVERSE X)),
```

the generalization heuristic would rewrite it to be:

```
(EQUAL (APPEND Y NIL) Y).
```

See [7] for the details.

The induction routine exploits the dualism between recursion and Burstall's structual induction, [3], to construct an induction argument. In its simplest form, the idea is that if the conjecture involves an application of some recursive function, F, and that application would recursively decompose the structure represented by some variable, X, then one should consider induction on the structure of X.

The intuitive justification is as follows: The induction principle requires proof that if the conjecture holds for the output of F applied to some value of X, then it holds for the output of F applied to some "larger" structure. In our implementation (but not Burstall's formulation), "larger" means simply that the structure can be obtained from X (and possibly other structures) by some fixed sequence of CONSes. Now by construction, an application of F in the conclusion is recursively

decomposing that larger structure which contains X. By analysis of the definition of F, we can choose that structure with X so placed within it to make one expansion with the definition of F produce an expression involving the (recursive) application of F to X. But since our induction hypothesis supplies information about the result of this very application, we can (possibly) use the hypothesis to simplify the conclusion.

Of course, to complete the inductive argument, sufficient "base cases" must also be established so that all possible structures can be obtained from the base cases and repeated applications of the sequence of CONSes used to construct the larger X.

Section 4 provides an example of an induction argument produced by this heuristic. Several full proofs are described in [2] and the heuristics are explained in detail in [7].

2. OUTLINE OF THE EXTENSION TO ITERATION

We will proceed as follows: We first introduce the LISP primitives allowing the definition of iterative programs. We then sketch a method of translating such iterative definitions into recursive programs that compute equivalent results "in the same way."

Then we demonstrate how the original induction heuristic fails to cope with such translations. Provided the conjecture to be proved is sufficiently general, the necessary extension to the induction heuristic is straightforward. This extension is explained and demonstrated.

We then turn our attention to the problem of generalizing the conjecture to enable the new induction heuristic to apply. The basic problem is generalizing from a conjecture about the state of the computation after one iteration to a conjecture about the state after an indefinite number of iterations. It is shown how a machine generated recursive function can be introduced to express the generalization for a particular class of iterative functions. It is then shown how the generalization is guaranteed to allow the induction step of the theorem prover's next induction argument to succeed. This will leave only the establishment of an "easier" base case -- with the caveat that the generalized conjecture might not be a theorem because the base case will be false.

We then justify the rather unexpected introduction of the generalization. This is done in two ways: an intuitive explanation of what the generalization is doing in terms of the state of the iterative computation, and a demonstration that the method is equivalent to replacing the iterative functions by less obvious but better behaved recursive functions at a point in the proof that is guaranteed to allow a similifying generalization (of the old type).

An example of the method at work is then exhibited.

The paper concludes with a brief comparison of this method with the work of other authors.

Two appendices list several iterative functions and some of their properties automatically proved by the extended program.

3. THE ITERATIVE PRIMITIVES

We will assume that the reader is familiar with the behavior of the LISP iterative primitives PROG, SETQ, GO, and RETURN. Consider the following iterative LISP program, called REVERSE:[3], which reverses its argument by CONSing the elements onto an initially empty list:

```
(REVERSE: (LAMBDA (X)
                  (PROG (Y)
                        (SETQ Y NIL)
                     LP (COND ((NULL X) (RETURN Y)))
                        (SETQ Y (CONS (CAR X) Y))
                        (SETQ X (CDR X))
                        (GO LP)))).
```

How can the techniques of the Boyer-Moore theorem prover be applied to such a function? Since both the notion and the notation of recursion are crucial to the discovery and representation of an induction argument by the Boyer-Moore theorem prover, it is a logical step to view this definition as an abbreviation for the following recursive definition:

```
(REVERSE: (LAMBDA (X) (REVERSE:LP X NIL))),
```

where REVERSE:LP is defined as:

```
(REVERSE:LP (LAMBDA (X Y)
                    (COND ((NULL X) Y)
                          (T (REVERSE:LP (CDR X)
                                         (CONS (CAR X) Y)))))).
```

This way of introducing PROGs into the system was described in [7]. It should be

3
In this paper, all functions defined with the iterative primitives have names ending with the letter ":".

clear that the two definitions of REVERSE: (above) return the same result. Indeed, they calculate the result in the same way: In calculating (REVERSE: A) the values of X and Y at the nth arrival at LP in the iterative definition will be precisely the same as the values of X and Y at the nth call to REVERSE:LP from the recursive definition. From this assertion (which is easily proved), it is clear that the recursive definition can be viewed as merely a notation for describing the state of the iterative computation. In particular, any recursive call of the form (REVERSE:LP A B) can be considered to denote the result returned by a computation that arrives at LP (in the iterative definition of REVERSE:) with X set to A and Y to B.

It is quite easy to make the translation from an arbitrary PROG to a collection of recursive functions. The method was first mentioned by McCarthy in [6]. Essentially, one generates a new recursive function for every label in the PROG. Each new function has as many arguments as there are formal parameters to the function, and locals to the PROG. One then accumulates the effects of the assignments down all branches. When a GO is encountered, it is translated into a call to the recursive function associated with the label to which control is being transferred. The arguments to the call are just the current values of the formals and locals.

Formally, we will consider forms involving the iterative primitives to be merely abbreviations expanded as sketched above. However, we shall choose to view this translation as a notation for packaging up the iteration so that we can talk about it. The rest of this paper deals with how one proves theorems about the functions produced by this translation. We do not dwell on the translation mechanism itself since it is fairly obvious and has been described in [6].

4. ACCUMULATORS

We call the Y-position argument of REVERSE:LP an *accumulator* because it is used to accumulate partial results during the recursion. In general, if a function modifies an argument during (some) recursive call but does not test the argument in the termination conditions, the program considers that argument to be an accumulator. Most functions produced by the translation mechanism described above use accumulators. The only exceptions are those functions which do not save partial results from one iteration to the next. For such functions the iterative definition cannot be distinquished by the theorem prover from the natural recursive definition. MEMBER, SUBSET, and many other boolean valued functions are good examples.

However, functions which use accumulators cause trouble for the induction heuristic described in the original papers. Consider the following theorem:

(1) (EQUAL (REVERSE:LP A B) (APPEND (REVERSE A) B)),

where APPEND and REVERSE are defined as in the original papers:

```
(APPEND (LAMBDA (X Y)
                (COND ((NULL X) Y)
                      (T (CONS (CAR X) (APPEND (CDR X) Y))))))
```

and

```
(REVERSE (LAMBDA (X)
                 (COND ((NULL X) NIL)
                       (T (APPEND (REVERSE (CDR X))
                                  (CONS (CAR X) NIL))))))).
```

Expression (1) states that the result of a computation which arrives at LP in the iterative definition of REVERSE: with X set to A and Y set to B is equivalent to the result of reversing A with the recursive function REVERSE, and then APPENDing

it to B. The sense in which (1) is a theorem is that it is claimed to be T for all possible values of A and B[4].

Consider how the induction heuristic described in [2] and [7] would attempt to prove (1). The heuristic chooses to induct upon the structure of A because that structure is being recursively decomposed by REVERSE and iteratively decomposed by REVERSE:LP (although in our representation of REVERSE:LP the theorem prover does not detect the difference between iteration and recursion). We say that "induction on A" has been chosen. Note that induction on B was not even considered since no function call in (1) recursively decomposes B.

The heuristic next discovers that both recursions on A recurse on the CDR of A. This means that the "larger" structure (to replace the A's in (1) to form the conclusion) should just be a single CONS with A in the CDR: (CONS A1 A), where A1 is a new Skolem constant. We are to supply a hypothesis by instantiating (1) for each of the substructures (of the larger structure) that is involved in a recursive call. This means we need a hypothesis about the CDR of the CONS above (i.e. about A). This is obtained by instantiating (1) by replacing A with A. (This is a trivial substitution simply because we chose to use A as the CDR of that CONS to make it so.) Since no recursion is interested in the CAR component of the CONS, no hypothesis about A1 is necessary. Finally, the base case is simply that instance of (1) where A is not the output of any CONS, which is to say, the case where A is simply NIL.

Therefore, the entire inductive argument is represented by the formula:

[4]*Informally,* A *and* B *can be considered to be program variables whose values are unknown and may range over all possible objects (*NIL *and all of the trees one can construct with* CONS*). Formally,* A *and* B *are constants, called Skolem constants. See the discussion and proof of Herbrand's Theorem in [8].*

```
(AND (EQUAL (REVERSE:LP NIL B) (APPEND (REVERSE NIL) B))
     (IMPLIES (EQUAL (REVERSE:LP A B)
                     (APPEND (REVERSE A) B))
              (EQUAL (REVERSE:LP (CONS A1 A) B)
                     (APPEND (REVERSE (CONS A1 A)) B)))),
```

where A, A1, and B are Skolem constants and AND and IMPLIES are just functions defined in terms of COND.

The base case is trivial since both sides of the equality:

```
(EQUAL (REVERSE:LP NIL B) (APPEND (REVERSE NIL) B))
```

simplify to B.

Therefore, consider the induction step. The hypothesis of the IMPLIES above is just (1). The conclusion is:

```
(2)  (EQUAL (REVERSE:LP (CONS A1 A) B)
            (APPEND (REVERSE (CONS A1 A)) B)).
```

Simplifying (2) by applying the definitions of REVERSE:LP and REVERSE produces:

```
(3)  (EQUAL (REVERSE:LP A (CONS A1 B))
            (APPEND (APPEND (REVERSE A) (CONS A1 NIL)) B)).
```

Note that we are completely unable to use our inductive hypothesis, (1). Why? Because (1) makes a statement about the result computed after arrival at LP with X set to A and Y set to B, but the (evaluated) conclusion, (3), is a statement about the result computed after arrival at LP with X set to A and Y set to (CONS A1 B).

5. INDUCTION REVISITED

Let us suppose we wish to prove some theorem:

$\mathbb{C}$((REVERSE:LP A B)),

where A and B are Skolem constants. This means we are trying to prove:

$\forall$ X (($\forall$ Y $\mathbb{C}$((REVERSE:LP X Y))).

Suppose we wish to induct upon X. The induction rule states that we must therefore prove the basis:

$\forall$ Y $\mathbb{C}$((REVERSE:LP NIL Y))

and we must prove the implication:

$\forall$ X (($\forall$ Y $\mathbb{C}$((REVERSE:LP X Y)))
$\rightarrow$
($\forall$ Y ($\forall$ Z $\mathbb{C}'$((REVERSE:LP (CONS Z X) Y))))),

where Z does not occur in $\mathbb{C}$ and $\mathbb{C}'$ is $\mathbb{C}$ with all the occurrences of X replaced by (CONS Z X). If we Skolemize this formula (see [9]) the conclusion becomes:

(4) $\mathbb{C}'$((REVERSE:LP (CONS A1 A) B)),

where A1, A and B are Skolem constants: we must prove the theorem without knowing anything about A1, A, or B. However, the hypothesis becomes:

(5) $\mathbb{C}$((REVERSE:LP A y)),

where A is the same Skolem constant as in the conclusion, (4), but y is a free variable: we get to assume the theorem for that particular A, and for all possible y's.

With this in mind we can now rewrite the induction conclusion, (4), using the definition of REVERSE:LP and get:

(6) $\mathbb{C}'$((REVERSE:LP A (CONS A1 B))).

We can now instantiate the free variable, y, in the hypothesis, (5), with (CONS A1 B). The hypothesis thus becomes:

(7) ℂ((REVERSE:LP A (CONS A1 B))),

so that the REVERSE:LP term in the rewritten conclusion, (6), matches the REVERSE:LP term in the instantiated hypothesis, (7), increasing the chances that we can use the hypothesis in the usual ways. (At the very least, if simplification and fertilization fail to use the hypothesis, the generalization heuristic described in Section 1 will replace the common REVERSE:LP expressions in the hypothesis and conclusion by a new variable, eliminating that function from the conjecture.)

Rather than provide a hypothesis with free variables in it, it is nicer to assume only those instances which one expects to need. This is because the interaction of the quantifiers becomes increasingly more complicated with subsequent inductions. Since we supply X with a CONS in the induction conclusion to allow it to "recurse back down" to the induction hypothesis, we should let y, in the hypothesis, be (CONS A1 B), knowing that is what the value of Y will be after the recursion.

The general rule we wish to add to our induction mechanism concerns only the generation of inductive hypotheses. It states that if we are doing induction on a structure which is controlling the accumulation of results in some argument position of some iterative function, and if that accumulator argument position is occupied by a Skolem constant other than one we are inducting upon, then, when forming the hypothesis for the recursive call of that function, replace that Skolem constant by the value forced into that argument position by a single step of the function.

6. AN EXAMPLE

With this in mind, let us reconsider the proof of (1):

```
(1)  (EQUAL (REVERSE:LP A B) (APPEND (REVERSE A) B)).
```

Again, we induct on A and again the NIL case is trivial. The induction conclusion is still (2):

```
(2)  (EQUAL (REVERSE:LP (CONS A1 A) B)
            (APPEND (REVERSE (CONS A1 A)) B)).
```

However, according to our new induction heuristic, the hypothesis is the instance of:

```
(8)   (EQUAL (REVERSE:LP A y) (APPEND (REVERSE A) y))
```

obtained by letting y be the value of the Y-accumulator of REVERSE:LP one step after X is (CONS A1 A) and Y is B. This just yields (9) as our induction hypothesis:

```
(9)  (EQUAL (REVERSE:LP A (CONS A1 B))
            (APPEND (REVERSE A) (CONS A1 B))).
```

Note that our hypothesis, (9), says: the result after arrival at LP with X set to A and Y set to (CONS A1 B) is equal to some expression. Our conclusion, (2), says: the result after arrival at LP with X set to (CONS A1 A) and Y set to B is equal to some (other) expression. That this is a sound induction argument is assured by the arguments in Section 5. That this is a useful induction argument is immediately apparent. (2) symbolically evaluates to (3):

```
(3)  (EQUAL (REVERSE:LP A (CONS A1 B))
            (APPEND (APPEND (REVERSE A) (CONS A1 NIL)) B)).
```

We can now use our induction hypothesis, (9), because the term (REVERSE:LP A

(CONS A1 B)) occurs as the left-hand side of the equality in (9) and as the left-hand side of the conclusion (3). Fertilization replaces the left-hand side of (3) with the right-hand side of (9), and then "throws away" the hypothesis, (9), producing:

```
(10)  (EQUAL (APPEND (REVERSE A) (CONS A1 B))
             (APPEND (APPEND (REVERSE A) (CONS A1 NIL)) B)).
```

Then the common subterm generalization heuristic described in Section 1 replaces the common subterm (REVERSE A), in (10), by the new Skolem constant C:

```
(EQUAL (APPEND C (CONS A1 B))
       (APPEND (APPEND C (CONS A1 NIL)) B)).
```

This expression is just a trivial lemma about APPEND, and can be easily proved by an induction on C. Thus, with a minor addition to the induction heuristic, we have proved the equality between an iterative form and a recursive one.

7. THE NEED FOR GENERALIZATION

In the example above, the induction step was proving that (2) follows from an instance, (9), of a valid induction hypothesis, (8). It is extremely important that we realize that y in (8) may be instantiated at will. Furthermore, it is important that we choose the correct instantiation. Consider the following description of the induction step above.

We decide to induct on A because we realize that A is being decomposed by the iteration in REVERSE:. We are required by the induction conclusion to show that the result computed after arrival at LP with X set to (CONS A1 A) and Y set to B is equivalent to some recursive expression involving (CONS A1 A) and B. As a

hypothesis, we may assume that the result of a computation that arrives at LP with X set to A and Y set to any value we choose is equivalent to some expression involving A and the chosen Y. Note that if (CONS A1 A) is the value of X at the nth arrival at LP, then A is the value at the n+1st arrival. This is why we induct on A. If, for a hypothesis, we choose the value of Y which is consistent with this view, that is, if we choose the value of Y which exists at the n+1st step, then the hypothesis will concern the computation state precisely one iteration after that described by the conclusion.

Two things should be noted regarding this induction argument. The first is that we are able to state the argument in terms of the ultimate result computed by the loop. This is a natural product of our notation but is not possible in the usual Floyd assertion method. For this reason the inductive arguments used by the program do not resemble those produced for the assertion method of program verification.

The second point is that the induction we are doing procedes in the direction opposite that of Floyd: we are assuming a property to hold at the n+1st step, and showing that it holds at the nth step. This sounds strange only because we are counting the steps from the initial entry into the loop. If we count forward to the termination of the loop, our hypothesis concerns the state existent k steps from termination, and our conclusion concerns the state k+1 steps from termination.

Of course, this is just the usual direction of a structural induction argument. Which way we count is unimportant, so we will count from the initial entry into the loop (in the Floyd way) and expect the reader not to become confused about which way the induction is leading.

What is important is that if we can interpret the conclusion as concerning the output of the nth step, then we should try to provide a hypothesis that concerns the state at the n+1st step.

To do this the accumulator terms must contain variables which allow the proper instantiations. In particular, we cannot do it if the accumulator positions are occupied by constants in the statement of the theorem. But this is a very common situation, since the accumulators arising from PROG locals are always assigned an initial value upon entry to the PROG. Thus, a theorem like:

```
(EQUAL (REVERSE: A) (REVERSE A)),
```

immediately becomes:

```
(EQUAL (REVERSE:LP A NIL) (REVERSE A)),
```

since REVERSE: assigns NIL as the initial value of Y, and we cannot instantiate the accumulator (the NIL in the REVERSE:LP term) as required by the induction heuristic since it is not a free variable.

We must discover a way to generalize the accumulator positions so that we can instantiate the hypothesis to provide information about the state of the iterative computation produced by the conclusion.

8. GENERALIZING ACCUMULATORS

The original generalization heuristic replaces subterms which occur in both sides of equalities or implications by new Skolem constants. The justification for the heuristic is that it is not so important what the value of such a term is as that the same value plays a role on both sides of the statement.

Replacing a term by a variable on only one side of, say, an equality is "dangerous" because it is possible to instantiate that variable and change one side but not the other. But if the generalized term occurs on both sides, then any changes in the value affect both sides.

By these arguments it is natural to require that an accumulator argument be replaced by a more general expression only when another accumulator argument, used in the same way, can be found on the other side of an equality or implication, to absorb the effect of the generalization. We say that these two accumulator values are *matched*. Requiring that both terms be matching accumulators is a way of insuring that changes in the value of the accumulator effect both sides in the same way.

It is actually very easy to get matched accumulators on each side of an equality or implication. This is done by induction.

For example, if the term (REVERSE:LP A NIL) occurs in the hypothesis of an induction on A, then the term (REVERSE:LP A (CONS A1 NIL)) will occur in the conclusion. We therefore match the NIL in the REVERSE:LP term in the hypothesis with the (CONS A1 NIL) in the REVERSE:LP term in the conclusion.

In general, if we are trying to prove some theorem involving an accumulator-using function, and we cannot instantiate the accumulator as we describe above, we merely do the induction without the desired hypothesis (i.e. use the hypothesis generated by the original heuristic). The resulting implication (after simplification) will have the initial value of the accumulator in the hypothesis, and the subsequent value of the accumulator in the conclusion. (If the theorem is an equality, a fertilization may take place, with the result that the matching accumulator values are on opposite

sides of an equality.) We have therefore solved both the problem of obtaining a form of the theorem with accumulators on both sides, and the problem of how to match multiple accumulators (match an accumulator in the hypothesis with the value it obtained in the conclusion).

This leaves us with the question: How do we create a reasonable generalization? A reasonable generalization is an expression which has the original expression as an instance, which we have reason to believe is in fact a theorem, and which is easier to prove than the original expression.

We clearly cannot replace the matched accumulators by some new Skolem constant, simply because the accumulators are not identical: the result would not be a sound generalization. The best we can do is replace each accumulator by some expression containing a free variable, and to define the expression in such a way that for some value of that variable the expression takes on the value of the accumulator it replaced. But the whole motivation behind the generalization is to allow the induction heuristic to instantiate the accumulator positions in the hypothesis in such a way that they take on the values they will have in the conclusion. This means that for some value of the free variable (the one used in the instantiated hypothesis) the expression must take on the value of the accumulator in the conclusion. However, the value of the accumulator in the conclusion will just be any value the accumulator can take on in one iteration from any of the possible values of the expression in the uninstantiated hypothesis. Thus, the expression must be capable of being instantiated to provide each of the successive values of the accumulator.

It turns out that the generalization heuristic will arrange this by writing a new recursive function, based on the behavior of the accumulator around the loop.

Furthermore, it turns out that this function need not necessarily capture all of the structure of the accumulator.

The generalization heuristic about to be described takes the theorem to be proved and two indicated occurrences of accumulator values, AC_1 and AC_2, which have been chosen for generalization[5], and produces a new theorem to be proved, by possibly renaming selected occurrences of some variables and by replacing the relevant occurrences of AC_1 and AC_2 by new expressions E_1 and E_2 respectively. By the nature of E_1 and E_2 the result is guaranteed to be more general. Furthermore, the generalization has the following startling property: The induction step, that is the implication from inductive hypothesis to inductive conclusion, for an induction on the variables controlling the loop using these accumulators, is guaranteed to be true and proveable by simplification! In fact, the induction hypothesis and the induction conclusion will usually evaluate to identical expressions! The danger is that the basis case(s) for the induction may not be true. That is, the generalization may not be a theorem because the basis breaks down.

We will describe this heuristc for a class of iterative functions somewhat more restricted than necessary. This is done for simplicity. Certain extensions to this class are obvious and are mentioned.

Let the function F: have the following defining form:

5
Actually the method applies to any number of matched accumulators. We use just two here for illustration, since our description of accumulator matching just picks pairs. The matching algorithm actually used may group more than two accumulator values together as matches, on the same ancestral grounds described. The actual generalization heuristic accepts groups of values, rather than just pairs, and the heuristic about to be described works perfectly well for such groups.

```
(F: (LAMBDA (X)
            (PROG (Y)
                  (SETQ Y NIL)
               LP (COND ((NULL X) (RETURN Y)))
                  (SETQ Y (H X Y))
                  (SETQ X (CDR X))
                  (GO LP)))),
```

where H is an arbitrary function of two arguments. Actually, since H is arbitrary, Y may not be modified in the same way each time around the loop. Furthermore, F: may have more than one formal argument, there may be more than one local to the PROG, the locals need not be initialized to NIL, the exit condition may be other than a simple NULL check (but may not involve the accumulator), some arbitrary function may be applied to the value of the accumulator when the RETURN is executed, and X may be decomposed in more complicated ways than merely CDRing. We will ignore these possibilities and leave it to the reader to convince himself that these introduce minor variations of the generalization heuristic.

Note that the recursive translation of F: is as follows:

```
(F: (LAMBDA (X) (F:LP X NIL))),
```

where F:LP is defined as:

```
(F:LP (LAMBDA (X Y)
              (COND ((NULL X) Y)
                    (T (F:LP (CDR X) (H X Y)))))).
```

Let the theorem to be proved be:

(11) $\mathbb{C}$((F:LP A AC_1) (F:LP A AC_2)),

where $\mathbb{C}$ is an arbitrary expression involving the two F:LP terms using the accumulators to be generalized.

The result of accumulator generalization applied to (11) will be a new expression,

(12) $\mathbb{C}$((F:LP X (E Z AC_1)) (F:LP X (E Z AC_2))),

where E is a new, recursive function, written by the theorem prover from the definition of F:LP, and X and Z are new Skolem constants, occuring only in the F:LP terms, and only as explicitly shown in (12). This expression, (12), will have the properties guaranteed above: it will be a generalization of (11) and the inductive step for an induction on X will be trivially true.

This is accomplished by defining the E, for this F:LP[6], to be:

```
(E (LAMBDA (Z Y)
           (COND ((NULL Z) Y)
                 (T (H (CAR Z) (E (CDR Z) Y))))))),
```

where the H used in this definition is that used to build the accumulator in F:LP. Note that E does not depend upon the AC_i. As we indicated above, E is capable of taking on any value the accumulator can have after any computation sequence because it can be given the initial value of the accumulator (Y) and a list which contains the entire succession of values of the control parameter for the particular computation sequence desired (Z).

Given this definition of E, (12) is clearly a generalization of (11) because when X is A and Z is NIL the evaluation of the E terms in (12) makes that expression identical to (11).

6
In general there is an E *for each accumulator position. We call this* E *the "Y-accumulator function" for* F:LP.

Now consider the inductive step for an induction on X to prove (12). The conclusion will be:

(13) $\mathbb{C}$((F:LP (CONS X1 X) (E Z AC_1)) (F:LP (CONS X1 X) (E Z AC_2))).

This evaluates to:

(14) $\mathbb{C}$((F:LP X (H (CONS X1 X) (E Z AC_1)))
(F:LP X (H (CONS X1 X) (E Z AC_2)))).

Now a valid induction hypothesis is any instance of:

(15) $\mathbb{C}$((F:LP X (E z AC_1)) (F:LP X (E z AC_2))).

Because of the way E is defined we choose to instantiate z in (15) with (CONS (CONS X1 X) Z), where the X1, X, and Z are the same Skolem constants appearing in the conclusion, (13). Thus our hypothesis ((15) with z instantiated as above) is:

(16) $\mathbb{C}$((F:LP X (E (CONS (CONS X1 X) Z) AC_1))
(F:LP X (E (CONS (CONS X1 X) Z) AC_2))).

But because E may be evaluated one recursive step when its first argument is a CONS, as above, (16) becomes:

(17) $\mathbb{C}$((F:LP X (H (CONS X1 X) (E Z AC_1)))
(F:LP X (H (CONS X1 X) (E Z AC_2)))),

by evaluation.

But the instantiated evaluated hypothesis, (17), is identical to the evaluated conclusion, (14)! Thus, the induction step is straightforward.

The danger of this generalization is that the NIL case for the induction above may not hold. Note that the NIL case is simply:

(18) $\mathbb{C}$((E Z AC_1) (E Z AC_2)).

It happens that there are a great many theorems for which this NIL case is in fact true and proveable with an additional induction[7].

9. JUSTIFICATION OF ACCUMULATOR GENERALIZATION

The arguments above are just so much syntactic magic. The real questions are: What is the relationship between F:LP and E? Why does the induction step go through so easily when E is introduced? Why should the NIL case, (18), be true, and if true, why should it be easier to prove than (11), which it greatly resembles?

The relationship between F:LP and E is clearly stated: given the proper Z, (E Z Y) is equivalent to (F:LP X Y). Let Z and X be lists of the same length, let the successive CDRs of Z be Z_0, Z_1, etc., and let the successive CDRs of X be X_0, X_1, etc. Then (E Z Y) computes:

(H (CAR Z_0) (H (CAR Z_1) ... (H (CAR Z_n) Y)...)),

while (F:LP X Y) computes:

(H X_n (H X_{n-1} ... (H X_0 Y)...)).

Thus, if we let (CAR Z_i) be X_{n-i}, we see that (E Z Y) computes (F:LP X Y).

7
Rod Burstall, in a private communication, has pointed out that accumulator generalization could be viewed simply as a proof rule that says that a possible way of proving a theorem like (11) *is to prove* (18) *instead.*

In general, Z is a list of the successive values of X as seen by H during the iteration in F:LP, in reverse order.

In fact, if F:REV computes a list of the successive values of the X in F:LP as a function of the initial value of X, then:

```
(F:LP X Y) = (E (F:REV X) Y).
```

But given this relationship, we could obtain (18) more or less directly from (11) by simply replacing the F:LP expressions in (11) by the equivalent E expressions and then generalizing away the F:REV expressions. We know we would generalize those expressions simply because of the criteria we used for selecting matching accumulators in the first place.

Thus, the whole issue of accumulator generalization can be recast simply as replacing the accumulator-using functions by more tractable, if less obvious, functions and then generalizing in the old way. Indeed, the question now arises as to why we should introduce the accumulator-using functions at all, rather than just use the equivalent E expressions from the beginning.

It is possible to show that introducing E expressions from the beginning is formally equivalent to the method described here. However it is somewhat clumsier due to the presence of F:REV. In addition, we know that if we stay with the original formulation of the problem, the technique in Section 5 will let us avoid the introduction of both E and F:REV altogether in the cases where the accumulator can be instantiated. Finally of course, the view of E as capturing the generalized state of the accumulator is instructive and offers new insights into structural induction on loops.

If we do view the step from (11) to (18) as accumulator generalization, why does the induction step work? The induction conclusion is about the nth iteration. We know that the result of that iteration is the same as that of the n+1st iteration, provided the function does not terminate immediately after the nth iteration. If it terminates after the nth iteration we do not need our induction hypothesis at all. If it does not terminate, we get to make a hypothesis about the result of the loop when X is the same as it is in the n+1st iteration. If we generalize the state of the accumulator by introducing E and the new variable z we can choose z so that the value of the accumulator in the hypothesis is also that found at the n+1st iteration. Therefore, we have constructed precisely the hypothesis we need: The theorem holds for the result of the n+1st iteration, which we know is the result of the nth iteration.

Regardless of which way we view the step to (18), why should (18) be easier to prove than (11)? The reason is that (11) involves F:LP terms which use instantiated accumulators, while (18) involves E terms which are very well-behaved. By "well-behaved" is meant that a structural induction on the variable controlling the recursion in the E terms will cause the E terms in the conclusion to symbolically evaluate to expressions involving the E terms in the hypothesis. An F:LP term is not well-behaved unless the first argument is a Skolem constant and the accumulator contains a free variable which can be instantiated as described in Section 5. Note also that the expression (E (F:REV X) Y) is not well-behaved.

Because (E Z AC_i) is well-behaved, an induction on the CONS-structure of Z is effectively an induction on the computational form of the possible values of F:LP, starting with the accumulator set to AC_i.

In either view of the role of E it is conceptually convenient to observe certain conventions when writing the definition of E. Let A be the value of the control parameter, X, upon entry to the loop represented by F:LP (as it is in (11)). If H (in F:LP) can be expressed as a function of (CAR X) rather than of X, then E should be written using that definition of H. This allows us to think of Z as merely some succession of elements from A. Furthermore, if H does not involve X at all, then Z can be thought of as merely a number (the number of computation steps necessary for the iteration in F:LP to completely decompose A).

For example, these conventions mean that the generalized state of the accumulator in REVERSE: is obtained by appending some arbitrary list Z onto the front of its initial value. The generalized state of the accumulator in FACT: (see Appendix A) is obtained by multiplying its initial value by the product of the elements of an arbitrary list Z. The generalized state of the accumulator in EXP: (see Appendix A) is obtained by multiplying its initial value by C an arbitrary number of times (Z).

The reason we can often ignore the very strong relationship between Z and A, even though both may be involved in (18), is the same reason we can often safely replace common subterms after an induction: The process of induction often embeds sufficient information in the theorem itself to capture the necessary properties of the term being generalized. Let us illustrate this claim and the heuristics described by a final example.

10. AN EXAMPLE OF ACCUMULATOR GENERALIZATION

Suppose we have the following function for collecting all those elements of X which satisfy some (unspecified) property P:

```
(COLLECT: (LAMBDA (X)
                  (PROG (Y)
                        (SETQ Y NIL)
                     LP (COND ((NULL X) (RETURN Y))
                              ((P (CAR X))
                               (SETQ Y (CONS (CAR X) Y))))
                        (SETQ X (CDR X))
                        (GO LP)))).
```

This function translates to:

```
(COLLECT: (LAMBDA (X) (COLLECT:LP X NIL))),
```

where

```
(COLLECT:LP (LAMBDA (X Y)
                    (COND ((NULL X) Y)
                          ((P (CAR X)) (COLLECT:LP (CDR X)
                                                   (CONS (CAR X)
                                                         Y)))
                          (T (COLLECT:LP (CDR X) Y))))).
```

There are several interesting facts that one can prove about this function. One of them is that it returns a subset of X. For a formal proof we would have to define both SUBSET and its subroutine, MEMBER, since they are not primitive. However, we will only sketch a proof and will merely assume that the necessary properties of these functions can be proved (as indeed they can).

The theorem we wish to prove is stated simply as:

```
(SUBSET (COLLECT: A) A),
```

which is equivalent to:

```
(19) (SUBSET (COLLECT:LP A NIL) A).
```

Since the accumulator is set to NIL we cannot use our new induction heuristic at the moment. However, we proceed by inducting upon A. The NIL case is trivial

because (SUBSET NIL NIL) is T. We now let (19) be our induction hypothesis, and try to prove:

(20) (SUBSET (COLLECT:LP (CONS A1 A) NIL) (CONS A1 A)).

If A1 does not satisfy P then we must show that (19) implies

(SUBSET (COLLECT:LP A NIL) (CONS A1 A)).

This implication is just an instance of the easily proved lemma:

(IMPLIES (SUBSET X Y) (SUBSET X (CONS Z Y))).

Therefore, let us consider the other case, where A1 satisfies P. Under this assumption the induction conclusion, (20) becomes:

(21) (SUBSET (COLLECT:LP A (CONS A1 NIL)) (CONS A1 A)).

We find our hypothesis is not useful, since it concerns the case where the accumulator is NIL and the conclusion has the accumulator set to (CONS A1 NIL). However, the theorem to be proved is an implication, with (19) as the hypothesis and (21) as the conclusion:

```
(22) (IMPLIES (SUBSET (COLLECT:LP A NIL) A)
              (SUBSET (COLLECT:LP A (CONS A1 NIL)) (CONS A1 A))).
```

This expression has matched accumulators on each side of an implication. This is not unexpected of course, since we arrived at (22) via an induction, as explained above. We therefore wish to generalize the NIL in the hypothesis, and the (CONS A1 NIL) in the conclusion.

We must now generate the Y-accumulator function (E) for COLLECT:LP. First we identify an H in the definition of COLLECT:LP:

```
(H (LAMBDA (X1 Y)
           (COND ((P X1) (CONS X1 Y)) (T Y)))).
```

In selecting this definition of H we took advantage of the fact that it could be expressed as a function of (CAR X).

The Y-accumulator function (E) for COLLECT:LP is obtained by simply following the pattern for E, using the H defined above, and then simplifying by replacing H with its definition. This process yields:

```
(COLLECT:LP' (LAMBDA (Z Y)
                     (COND ((NULL Z) Y)
                           ((P (CAR Z))
                            (CONS (CAR Z)
                                  (COLLECT:LP' (CDR Z) Y)))
                           (T (COLLECT:LP' (CDR Z) Y))))).
```

Because we treated H as a function of (CAR X), we know that Z should be thought of as a sequence of elements from X. For strict equality between COLLECT:LP and COLLECT:LP', Z should be the reverse of X. However, we will ignore these restrictions.

Thus, using COLLECT:LP' to generalize the state of the accumulator implies that its general state is taken to be obtained by appending to it an arbitrary collection of elements satisfying P.

If we wish to illustrate accumulator generalization (rather than effectively stepping directly from (11) to (18)), the generalization we introduce looks fairly hideous:

```
(23) (IMPLIES (SUBSET (COLLECT:LP X (COLLECT:LP' z NIL)) A)
              (SUBSET (COLLECT:LP X (COLLECT:LP' z (CONS A1 NIL)))
                      (CONS A1 A))).
```

This is obtained from (22) in the same way that (12) is obtained from (11): The two accumulator values, NIL and (CONS A1 NIL), in (22) are embedded in COLLECT:LP' expressions, and two of the occurrences of the variable controlling the iteration for those accumulators, A, are replaced by the new variable X. Note that there are other occurrences of A which are not replaced by X.

Of course, (23) is just momentarily hideous, since we know that an induction on X, with the proper instantiation of z in the hypothesis, is trivial and will leave us with the base case only. This may not be as apparent as it was in the F:LP example. However, we will leave it to the reader to convince himself that if the induction hypothesis is (23) with z replaced by (CONS X1 Z), and the induction conclusion is (23) with X replaced by (CONS X1 X) and z replaced by Z, then the hypothesis and conclusion evaluate to precisely the same expression. Therefore, there is actually no need to introduce (23), since we are assured the induction step will go through. Instead, we can merely consider the base case, obtained by letting X be NIL in (23):

```
(24) (IMPLIES (SUBSET (COLLECT:LP' Z NIL) A)
              (SUBSET (COLLECT:LP' Z (CONS A1 NIL)) (CONS A1 A))).
```

We are now in a position to illustrate the claim that the base case often holds, even though we ignore the complicated restrictions on Z. Recall that the only condition under which the COLLECT:LP' expressions in (24) are equivalent to the COLLECT:LP expressions in (22) is when Z is the reverse of A. However, we are now claiming, effectively, that we can replace the COLLECT:LP expressions in (22) by the COLLECT:LP' expressions in (24) and still have a theorem, without even linking the new Z to the old A still in (24).

Hopefully the reader is convinced that (24) is indeed true. It says that if one

recursively collects the elements satisfying P in some unrestricted list, Z, and appends them to (CONS A1 NIL), then the result is a SUBSET of (CONS A1 A), provided those elements in Z which satisfy P form a SUBSET of A. It is proveable by a single induction on Z. But why is it true, since we have ignored a very tight restriction on the relationship between Z and A? The answer is that we arrived at (22), the expression from which we produced (24), by induction on A. That induction embedded sufficient information about the relation between Z and A to make the theorem go through. In particular, it gave us the condition that (COLLECT:LP' Z NIL) is a SUBSET of A: the elements of Z satisfying P form a SUBSET of A.

11. CAPABILITIES AND FURTHER WORK

The heuristics described here produce automatic proofs of such theorems as the equivalence of the iterative and recursive definitions of factorial and exponentiation, various recursive relationships involving iterative methods of collecting and counting, and the correctness of a simple iterative list sorting function. The correctness result is proved as two theorems, the first stating that the output of the SORT: function is ORDERED and the second stating that the output is a permutation of the input. The permutation result is stated simply as:

```
(EQUAL (COUNT A (SORT: B)) (COUNT A B)),
```

where COUNT counts the number of occurrences of its first argument in the list represented by its second.

Appendix A lists some sample iterative function definitions. Appendix B lists a variety of theorems the program has proved about these functions.

Of course, the heuristics described are not complete. The accumulator generalization heuristic sometimes produces expressions which are not theorems, because too much information is lost regarding the relationship between the old and new variables. A simple example can be obtained by writing the most obvious iterative function for generating a list of the numbers from 1 to N and then proving that the output is always ORDERED. The generalization produced effectively states that if Z is ORDERED then the the result of appending N as the last element is also ORDERED. This is true only if Z is known to contain no element greater than N, which is missed by the generalization. An extension of the machine generated "type functions" of [7] is being considered as a means of restricting the generalizations.

The translation of nested iteration is straightforward, but proving facts about the translations is somewhat more difficult. This is being studied with Ben Wegbreit.

12. RELATED WORK

There are three areas of related work. The first is the program optimization work of Darlington and Burstall, [4]. Under certain conditions on H it is easy to write a well-behaved definition of F:LP which is much like E. It is clear that if the theorem prover actually checked for these conditions and translated F:LP as described in [4], many proofs would be simpler.

The second area is that of the automatic generation of induction assertions for program verifiers using the Floyd method. Two methods studied were those of Wegbreit, [10], and Katz and Manna, [5]. Of course, the central problem is to generate some assertion about the general state of the computation within a loop, and to insure that the assertion is relevant to the overall goal of proving some

theorem about the output. It is difficult to compare our method and theirs because we express assertions in terms of the ultimate output of the loop and we allow assertions concerning the output of two or more distinct computation sequences through the loop (as in (23)). Our method resembles the "top-down" approach (in the terminology of [5]) in that the generalization is produced from the theorem being proved. This insures its relevancy. However, our method also resembles the "bottom-up" approach since the program analyzes the behavior of the loop to extract the general description of the state of the accumulator. The relationship between the Boyer-Moore theorem prover and the automatic generation of Floyd assertions is being explored with Ben Wegbreit.

The final related area is the similar but independent work by Aubin, [1], who is using the original implementation of the Boyer-Moore theorem prover at the University of Edinburgh. Aubin's suggested generalization method involves using look-ahead to discover how our matched accumulators fail to unify. He proposes to generalize the original theorem, introducing a free variable as we do, and only then proceed with his first induction.

The difficulty with this approach, shared by [10] and [5], is illustrated by considering how one might generalize the NIL in (EQUAL (REVERSE:LP A NIL) (REVERSE A)). Clearly any generalization of that term must somehow be reflected on the other side of the equality. To do this one must "understand" both how the accumulator is being modified and how it relates to the rest of the theorem.

We avoid the problem by waiting until induction has introduced balanced accumulators. We then have a relationship between two distinct computations through the loop, starting at different accumulator values. We then merely try to

show that this relationship holds for the generalized state of the accumulators. Thus we have decoupled the problem of understanding how the accumulators are used from the problem of understanding their relationship with the rest of the theorem.

13. REMARKS ON THE IMPLEMENTATION

The theorem prover is implemented in INTERLISP-10 at Xerox PARC. Unlike the original version, the induction formula is now set up without evaluating the theorem and looking at the failures. Instead, functions are carefully analyzed when they are defined, and this analysis is stored and used to generate inductive arguments.

The new version of the program also makes extensive use of previously proved theorems as lemmas. The use of lemmas allows the theorem prover to handle more difficult programs since the properties of subroutines can be verified first. When a new property arises the program can still fall back on the recursive definitions to try to establish the result inductively.

The new version of the program is several times faster than the original. The average proof (from a sample of about 100 theorems proved routinely after program modifications) requires about 6 seconds of cpu time. Roughly 95% of that time is spent in the simplification routine.

ACKNOWLEDGEMENTS

I would like to thank Bob Boyer and Ben Wegbreit for hours of clarifying conversations on the topic presented here. In addition, Raymond Aubin, Rod Burstall, Gordon Plotkin, and Richard Waldinger have also made several enlightening suggestions.

REFERENCES

[1] R. Aubin, "Generalization in Proofs of Simple Program Properties," unpublished memo, Department of Artificial Intelligence, University of Edinburgh, 1974.

[2] R. S. Boyer and J S. Moore, "Proving Theorems About LISP Functions," *Journal of the ACM*, January, 1975.

[3] R. M. Burstall, "Proving Properties of Programs by Structural Induction," *Computer Journal*, pp. 41-48, Vol. 12, 1969.

[4] J. Darlington and R. M. Burstall, "A System which Automatically Improves Programs," *Proc. of the Third IJCAI*, pp. 537-542, 1973.

[5] S. M. Katz and Z. Manna, "A Heuristic Approach to Program Verification," *Proc. of the Third IJCAI*, pp. 500-512, 1973.

[6] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *Communications of the ACM*, Vol. 3, April 1960.

[7] J S. Moore, "Computational Logic: Structure Sharing and Proving Program Properties," Ph.D. Thesis, University of Edinburgh, 1973.

[8] J. R. Shoenfield, *Mathematical Logic*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1967.

[9] H. Wang, "Toward Mechanical Mathematics," *IBM Journal*, pp. 2-22, January 1960.

[10] B. Wegbreit, "Heuristic Methods of Mechanically Deriving Inductive Assertions," *Proc. of the Third IJCAI*, pp. 524-536, 1973.

APPENDIX A - FUNCTION DEFINITIONS

The functions defined below are involved in the theorems exhibited in APPENDIX B. In the case where a function is used but not defined, the definition is the same as in [2].

```
(EXP (LAMBDA (C X)
       (COND ((ZEROP X) 1)
             (T (TIMES C (EXP C (SUB1 X)))))))

(EXP: (LAMBDA (C X)
        (PROG (Y)
              (SETQ Y 1)
          LP  (COND ((ZEROP X) (RETURN Y)))
              (SETQ Y (TIMES C Y))
              (SETQ X (SUB1 X))
              (GO LP))))

(FACT (LAMBDA (X)
       (COND ((ZEROP X) 1)
             (T (TIMES X (FACT (SUB1 X)))))))

(FACT: (LAMBDA (X)
         (PROG (Y)
               (SETQ Y 1)
           LP  (COND ((ZEROP X) (RETURN Y)))
               (SETQ Y (TIMES X Y))
               (SETQ X (SUB1 X))
               (GO LP))))

(UNION: (LAMBDA (X Y)
          (PROG NIL
            LP  (COND ((NULL X) (RETURN Y))
                      ((NOT (MEMBER (CAR X) Y))
                       (SETQ Y (CONS (CAR X) Y))))
                (SETQ X (CDR X))
                (GO LP))))

(INTERSECT: (LAMBDA (X Y)
              (PROG (Z)
                    (SETQ Z NIL)
                LP  (COND ((NULL X) (RETURN Z))
```

```
                    ((MEMBER (CAR X) Y)
                     (SETQ Z (CONS (CAR X) Z))))
                (SETQ X (CDR X))
                (GO LP))))

(SORT: (LAMBDA (X)
         (PROG (Y)
               (SETQ Y NIL)
           LP  (COND ((NULL X) (RETURN Y)))
               (SETQ Y (ADDTOLIST (CAR X) Y))
               (SETQ X (CDR X))
               (GO LP))))
```

APPENDIX B - SAMPLE THEOREMS PROVED

Below are some of the theorems proved by the extended program. In some cases the program appeals to lemmas which it has previously proved (such as the laws governing PLUS and TIMES). While this list of theorems is by no means complete, it is representative of the upper limits of the program's current competence.

```
(EQUAL (FACT: I) (FACT I))

(EQUAL (FACT:LP I J) (TIMES J (FACT I)))

(EQUAL (EXP: I J) (EXP I J))

(EQUAL (EXP:LP I J K) (TIMES K (EXP: I J)))

(EQUAL (EXP: I (PLUS J K)) (TIMES (EXP: I J) (EXP: I K)))

(EQUAL (EXP: I (TIMES J K)) (EXP: (EXP: I J) K))

(IMPLIES (MEMBER A B) (MEMBER A (UNION: B C)))

(IMPLIES (MEMBER A C) (MEMBER A (UNION: B C)))

(IFF (MEMBER A (UNION: B C)) (MEMBER A (UNION B C)))

(SUBSET A (UNION: A B))
```

```
(IMPLIES (AND (MEMBER A B) (MEMBER A C))
         (MEMBER A (INTERSECT: B C)))

(IFF (MEMBER A (INTERSECT: B C)) (MEMBER A (INTERSECT B C)))

(SUBSET (INTERSECT: A B) A)

(ORDERED (SORT: A))

(EQUAL (COUNT A (SORT: B)) (COUNT A B))
```

Note that these last two results establish that SORT: is correct: it produces ordered output and its output is a permutation of its input.